Robust Architectural Support for Transactional Memory in the Power Architecture

Harold W. Cain*
IBM Research
Yorktown Heights, NY, USA
tcain@qti.qualcomm.com

Brad Frey
IBM STG
Austin, TX, USA
bradf@us.ibm.com

Derek Williams
IBM STG
Austin, TX, USA
striker@us.ibm.com

Maged M. Michael
IBM Research
Yorktown Heights, NY, USA
magedm@us.ibm.com

Cathy May
IBM Research (retired)
Yorktown Heights, NY, USA
mayggg@gmail.com

Hung Le
IBM STG
Austin, TX, USA
hung@us.ibm.com

ABSTRACT

On the twentieth anniversary of the original publication [10], following ten years of intense activity in the research literature, hardware support for transactional memory (TM) has finally become a commercial reality, with HTM-enabled chips currently or soon-to-be available from many hardware vendors. In this paper we describe architectural support for TM added to a future version of the Power ISA™. Two imperatives drove the development: the desire to complement our weakly-consistent memory model with a more friendly interface to simplify the development and porting of multithreaded applications, and the need for robustness beyond that of some early implementations. In the process of commercializing the feature, we had to resolve some previously unexplored interactions between TM and existing features of the ISA, for example translation shootdown, interrupt handling, atomic read-modify-write primitives, and our weakly consistent memory model. We describe these interactions, the overall architecture, and discuss the motivation and rationale for our choices of architectural semantics, beyond what is typically found in reference manuals.

1. INTRODUCTION

This work describes the first architectural support for TM in the Power ISA. (The TM feature implemented in Blue Gene®/Q [1] contains no instruction set support.) Since the Power ISA is the basis for IBM® pSeries servers, whose strengths are robust and scalable performance in large system configurations (e.g. up to 256 cores / 1024 threads in current p795 systems, with 8 TB of DRAM), as well as strengths in RAS that differentiate it in the market, adding TM must not compromise any of these virtues. A robust system is one that is sturdy in construction, a trait that does not usually come to mind in respect to HTM systems. We structured TM to work in harmony with features that support the architecture's scalability. Our goal has been to provide a comprehensive programming environment including support for simple system calls and debug aids, while providing a robust (in the sense of "no surprises") execution environment with reasonably consistent performance and without unexpected transaction failures.[2] TM must be usable throughout the system stack: in hypervisors, operating systems, libraries, compilers, run-time systems, and applications. Further, TM must be easily exploited in existing code, as well as newly written software, driving the need to co-exist with nearly the full scope of the ISA's functionality. The transactional extensions start with a set of instructions for implementing strongly-isolated[17][3] transactions as well as a general mechanism for checkpointing and rollback of architectural state. It discourages unpredictable transaction behavior (e.g. aborting transactions on some but not all function calls [19]), while also providing support for identifying the source of transaction failures when the contents of the transaction are inconsistent with transactional semantics (e.g. code that performs a cache flush operation). One feature supporting these goals is *transactional suspension*, which implicitly occurs when handling interrupts, or can be explicitly invoked via a new suspend/resume instruction. This support is described in Section 3.

In the near term, the value of the TM feature is expected to come from lock elision[24] and to a lesser extent from speculative compilation techniques that are enabled by the checkpointing/rollback mechanism[23, 22, 28]. Over time, the ease of programming relative to the weakly consistent Power ISA memory model will seed the development and porting of multithreaded applications. (For example if multiple memory accesses can be wrapped in a transaction, there is no need to reason about the necessary memory barriers between those particular accesses.) The architecture has been

*The author is now employed by Qualcomm Research-Raleigh

[1]IBM, Power, Blue Gene, AIX, Power Architecture, and z/Architecture are registered trademarks of International Business Machines.

[2]We use the term *failure* to refer to any unsuccessful transaction, and reserve the term *abort* to refer to failures that are explicitly requested via transaction abort instructions.

[3]While Blundell et. al[17] use the term strongly atomic, we prefer and use the term strongly isolated.

ISCA '13 Tel-Aviv, Israel

tailored for these uses, while providing a base to support emerging language standards and eventually be extended for lock-free programming.

Another aspect of the inclusion of TM is its intersection with the Power ISA weak memory model. In addition to the interaction of TM with non-transactional weakly ordered code, since lock elision enables the transactionalization of arbitrary code, we must define correct and predictable semantics for the inclusion of memory barriers and atomic synchronization primitives (i.e. load-reserve/store conditional) within a transaction. We describe the intersection of TM and the Power ISA memory model in Section 4.

Although this paper summarizes the architectural support for HTM in one commercial architecture, and the design challenges and trade-offs involved, some of the features discussed have not been previously described in the literature, which we highlight here as novel contributions of this work:

- We define a suspended transactional mode of execution with deferred failure handling semantics, allowing for a sequential programming model that can run existing software (e.g. interrupt handlers) without fear of failure-induced control redirection. This mode is entered as the result of an interrupt or via a new suspend instruction, and is an important ingredient in the architecture's robustness. It is described in Section 3.
- We describe the potential for a loss of transactional atomicity in systems that implement hardware-based translation shootdowns (i.e. do not rely on interprocessor interrupts), motivating support for conflict detection mechanisms with translation shootdown.
- We describe various mechanisms, most importantly an integrated cumulative barrier effect on transaction committal, that are necessary to allow transactional and non-transactional accesses to interoperate in a sensible fashion. The interaction of TM and the Power ISA memory model is described in Section 4.
- We define a variant of transactions called *rollback-only transactions* that support the checkpointing and rollback of architectural state without the atomic features of transactions, to be used for software speculation.

We begin with a summary of the requirements that guided this architecture and a high-level overview in Section 2.

2. ARCHITECTURE: REQUIREMENTS AND OVERVIEW

The necessary support for our goals was reconciled against its expected implementation cost, which must be feasible across a broad class of possible implementations. We began with an assumption of a mode-based strongly isolated[17] best-effort HTM, with transactions initiated/committed by new instructions. One motivation for this design point, including the use of register checkpointing, is its compact means of starting and ending transactions with a single instruction each, as opposed to proposals that may require more software overhead such as saving/restoring registers in software, or alternative mechanisms for initiating, committing, and aborting transactions that may require memory-mapped accesses, software-visible logging, marking, or protection structures. Since TM will compete with alternate `larx/stcx` sequences, a compact mechanism is necessary to be competitive with those primitives. From this starting point, we also imposed the following implementation assumptions:

- We would not require support for maintaining more than one read or write set per transaction (i.e. no guaranteed support for intermediate rollback of nested transactions or multiple speculative versions of a line).
- There would be no guarantee of success for any transaction; all transactions must specify a failure handler allowing for non-transactional alternatives.
- Despite a desire for graceful handling of many rare events, there would be no requirement of support for transaction survival across context switches or paging of transactionally accessed data.
- When possible, the primitives themselves must be largely consistent with existing components of the Power ISA. Significant deviations were only permissible if absolutely necessary.
- The resulting architecture would allow for decoupled implementations of TM and existing larx/stcx atomic synchronization primitives.
- We placed no architectural limits on the transaction size an implementation must support, but encourage the possible support of large transactions by eliminating many causes of transaction failure. Implementations must have the flexibility to choose granule size for tracking transactional conflicts.

Support for TM was primarily motivated as an enabler of lock elision and speculative code generation techniques. Lock elision is a technique that can be implemented purely in hardware[24], purely in software [21], or using a combination of the two through transactional lock elision (TLE) [7], which is the approach we adopt. Lock elision requires the ability to execute transaction unaware code, as opposed to small carefully orchestrated transactions that may be constructed by a programmer by hand, for example when building a lock-free data structure. Since lock-based critical sections can make procedure calls that may end up distantly removed from the original critical section (potentially even in system libraries or the operating system), we would like to handle the execution of unpredictable code within a transaction, and in the event of failure in such a disparate location, convey enough information back to the originator of the transaction that a programmer can debug and understand the cause of such transaction failures.

Further, the ability to build a reliable, bug-free system is only possible with significant support for reliably debugging the system. Of concern in debugging transactional applications was a programmer's inability to "peek" inside failing transactions; in the absence of supporting mechanisms, a persistent transaction failure may be impossible to debug since all architectural updates are reverted on transaction failure. While a persistent failure may not be problem if transactions can always fall back on non-transactional code, once a user becomes accustomed to the benefits of transactional execution, subsequent changes to their code or the system's operating environment could lead to transaction failures causing those benefits to disappear. For this and other reasons outlined in Section 3, our TM system includes a mechanism for transactional suspension, which en-

ables the debugging of persistent failures by permitting non-transactional actions from within a transaction. It is from within this context that the following architecture emerged.

Transactional execution is controlled using two new modes of operation: Transactional and Suspended Transactional modes, which are controlled using a new 2-bit Transactional State (TS) in the machine state register (MSR).

2.1 Transactional Mode

Transactional mode is initiated via the execution of a `tbegin` instruction, and transactions are committed using a `tend` instruction. Transactions can fail due to a variety of causes, resulting in transactional stores being discarded as well as the rollback of all registers that are writable in user mode. Conceptually, this register state is checkpointed at transaction initiation in what we refer to as the *speculative register checkpoint*. Register checkpointing was motivated by a desire to support low-latency initiation of transactional execution in lock-elision and software speculation-enabled code, otherwise at the onset of speculation, software would be required to checkpoint any register state that may be modified, creating an initial performance deficit with respect to conventional code. Table 1 summarizes the new instructions comprising the transactional facility.

Transactions can be explicitly aborted by software using five new Transaction Abort instructions. The first, `tabort`, unconditionally aborts a transaction and copies eight bits from its register operand into a diagnostic register (TEXASR). The remaining four tabort instructions support comparison of two different operand widths (word and doubleword), conditionally causing transactional abort based on the comparison of two registers or a register and immediate. These conditional tabort instructions reduce pathlength (replacing a compare-branch-tabort sequence) in code with many transactional exits.

In addition, we include support for suspending and resuming transactions using the `tsr` instruction, reading the transaction mode and checking transaction validity using `tcheck`, and saving and restoring the checkpointed transactional register state on context switches via `treclaim` and `trechkpt`. These are described in more detail in Section 3.

An example transaction is shown in Figure 1. Most TM instructions, including `tbegin`, set a condition register as a side-effect, indicating the transactional state (abbreviated `MSR[TS]`) at the time that the instruction is executed. On successful initiation, the branch following `tbegin` will fall through to the transaction body. On failure, the branch will transfer control to a programmer-defined failure handler. In the overly simplified handler shown here, the logic simply tests a "persistent" hint bit located in the TEXASR, and if unset retries the transaction.

2.1.1 Registers for Failure Handling

We also add three new special-purpose registers to manage failure handling:

- Transaction Failure Handler Address Register (TFHAR): On the initiation of a transaction, records the address following `tbegin`, which is responsible for redirecting control flow to the transaction failure handler.
- Transaction Failure Instruction Address Register (TFIAR): Records the address of the instruction responsible for transaction failure, when available.

```
tmstart:
   tbegin 0      #TFHAR <= tbegin PC + 4
   beq- failure  #If failure, goto handler
   ...           #Transaction body
   tend          #Commit transaction
   ...
failure:
   mfspr r4, TEXASR   #Read diagnostic register
   andi. r4,r4,MASK   #Is persistent bit set?
   beq tmstart        #If not, retry, or use non-TM path
nontransactional_path:
   ...
```

Figure 1: Example transaction

- Transaction Exception and Status Register (TEXASR): Records a variety of information about the status of the current or most recently executed transaction, encoded in the fields shown in Table 2.

2.1.2 Transaction Failures

Transactions can fail for a number of reasons, as enumerated in the first half of Table 2. While many of these causes will be familiar to readers, we note the following causes which may not be obvious:

- Failures caused by detection of a suspend-mode access by the transactional thread that conflicts with its own read or write set, as further described in Section 2.2.
- Failures caused by conflicts with concurrent shootdown of translation entries, described in Section 3.3.
- Differentiation of failures caused by conflicts with transactional and non-transactional accesses.

On detection of failure, the act of failure processing is divided into two separate parts: *failure recording* and *failure handling*.

Failure recording is the process of recording information about the cause and circumstances of failure in SPRs associated with the transactional facility. This includes setting of failure cause bits defined in TEXASR, as well as failure context bits such as the privilege, suspend, and transaction level bits, and the setting of the TFIAR. Since some failure types may not be caused by a particular instruction (e.g. TM footprint overflow), or an associated PC may be difficult to capture (e.g. conflicts caused by stores), the TEXASR includes a TFIAR valid bit, indicating whether or not TFIAR is a precise address of the failure-causing instruction, or an approximation. Failure context bits are an aid to programmers to determine the failure's context, particularly in cases where the TFIAR does not contain a precise value. Failure recording is performed once per transaction that fails, despite cases where multiple failure conditions may simultaneously present themselves.

During failure handling, all updates to memory and speculative registers are rolled back, and control is transferred to the failure handler address, with a condition register (`CR0`) set to indicate that failure has occurred. The timing of failure handling is dependent on whether the machine is in the Transactional or Suspended Transactional mode at the time of detection. In Transactional mode, failure recording and handling occur immediately. In Suspended mode, failure recording occurs immediately on detection of the first failure, but in order to preserve the sequential semantics of

User-mode instructions	
`tbegin. R`	*Transaction begin.* Initiates transactional execution. The R parameter indicates this is a *roll-back only* transaction. Transaction initiation prevented in Suspend mode.
`tend. A`	*Transaction end.* Commits a transaction. When set, the A parameter forces commit of transaction regardless of current nesting level. Execution from Suspended mode is disallowed.
`tabort. RA`	*Transaction abort.* Unconditionally aborts a transaction. Lower byte of register parameter RA is copied into TEXASR to record cause of transaction failure. In Suspend mode, causes transaction failure recording, but not failure handling.
`tabortwc. TO, RA, RB` `tabortdc. TO, RA, RB`	*Transaction abort conditional.* Register operands RA and RB are compared using an operator specified by TO. Depending on the outcome of the comparison, the transaction is conditionally aborted. Word (w) and doubleword (d) variants are supported.
`tabortwci. TO, RA, SI` `tabortdci. TO, RA, SI`	*Transaction abort conditional immediate.* Similar to `tabort[w|d]c`, except compares register RA to a signed immediate operand.
`tsr. L`	*Transaction Suspend or Resume.* L parameter controls whether to suspend or resume. We refer to these two types as `tsuspend` and `tresume`.
`tcheck BF`	*Transaction Check.* Sets condition register indicating whether transaction failure has been detected, including whether prior memory operations are currently conflict-free. Can be used in Transactional and Suspended modes to check transaction validity.
Privileged and Hypervisor instructions	
`treclaim. RA`	*Transaction Reclaim.* Reclaims transactional facility for a new use, forcing failure for any active transaction. Lower byte from register operand RA copied to TEXASR, similar to tabort. Execution from Non-transactional mode is disallowed.
`trechkpt.`	*Transaction Recheckpoint.* Sets the transactional facility's register checkpoint to the current register state. Execution from Transactional or Suspended mode is disallowed.

Table 1: Summary of New Instructions

non-transactional execution, failure handling is deferred until transactional execution is resumed. We decouple failure recording and handling to allow for their separate occurrence when failure occurs during a suspended transaction. This decoupling also supports the precise recording of an initial failure cause; failure handling may be delayed for a variety of reasons, and for potentially a very long time (if a transaction is suspended). Supporting the early capture of failure cause separately from failure handling allows subsequent diagnosis of the original failure condition.

2.2 Suspended Transactional Mode

The Suspended transactional mode is explicitly entered with the execution of a `tsuspend` form of the `tsr` instruction during a transaction, the execution of a `trechkpt` instruction from non-transactional mode, or as a side-effect of an interrupt[4] while in the Transactional mode. Once suspended, transactional execution can be resumed by executing the `tresume` variant of the `tsr` instruction, or by execution of a return-from-interrupt instruction (e.g. `rfid`).

Suspended transactional execution is defined by the following semantics:

- Memory accesses are performed non-transactionally; they will be performed independently of the outcome of the transaction.
- The initiation of a new transaction is prevented.
- In the event of transaction failure, failure recording is performed, but failure handling is deferred until transactional execution is resumed.
- Until failure occurs, load instructions that access memory locations that were transactionally written by the same thread will return the transactionally written data. After failure is detected, but before failure handling is performed, such loads may return either the transactionally written data, or the current non-transactional contents of the accessed location.
- Store instructions that access memory locations that have been accessed transactionally (due to load or store) by the same thread will cause the transaction to fail.

These semantics provide a programming model identical to the non-transactional programming model so long as programmers follow these rules: they read transactionally written memory locations only as described in the next paragraph, they do not store to memory locations that have been transactionally read or written, and they do not expect transactions to be initiated successfully. Once in Suspended mode, the processor remains in Suspended mode following sequential execution semantics even in the presence of transaction failure; it is only after the transaction is resumed that failure handling takes place. This preservation of the sequential execution model allows for the execution of transaction-unaware code such as the hypervisor, operating system, or certain library calls (or portions thereof) in Suspended transactional mode.

Transactionally modified memory locations may be read, but on transaction failure those modifications may be discarded. Therefore, when a load is executed, it is uncertain whether the load returned the transactionally written value, or the current value present at the location. (The architecture guarantees that one or the other will be returned). A `tcheck` instruction can be used to check the validity of the current transaction, indicating that any prior loads to the transactional write set observed the transactionally written value. (Although it would have been nice to preserve the transactional stores until the end of the suspend-mode block, that would require saving/restoring the store footprint across context switches, a prohibitive implementation cost.) If a suspended transaction stores to a location in the transactional read or write set, transaction failure is incurred in order to preserve transactional semantics. (Such failure is handled when the transaction is resumed.)

The final difference between Non-transactional and Suspend mode execution is that any attempt to initiate a new transaction during suspend-mode execution is denied, with

[4]In the Power ISA, all system interactions occur via interrupts, i.e. we use this generic mechanism for system calls, synchronous exceptions, and asynchronous exceptions, so discussion of interrupts is applicable to all of these types of events.

Failure Cause Bits	
Field	*Description*
Disallowed	Set when failure is caused by a disallowed instruction or access type.
Nesting overflow	Set when failure is caused by exceeding the maximum transaction level.
Footprint overflow	Set when failure is caused by overflowing the capacity of the transactional footprint.
Non-transactional conflict	Set when failure is caused by a conflict with a non-transactional access.
Transactional conflict	Set when failure is caused by a conflict with a transactional access.
Self-induced conflict	Set when failure is caused by a self-induced conflict. (Described in Section 2.2.)
Translation invalidation conflict	Set when failure is caused by conflict with a translation shootdown. (Described in detail in Section 3.3)
Implementation-specific	Set when failure is caused by some other implementation-specific reason. Such failures must not be persistent.
Instruction fetch conflict	Set when failure is self-induced due to an ifetch that conflicts with a transactionally written location.
Abort	Set when failure is caused by the execution of an unconditional transaction abort or transaction reclaim instruction, in which case the Failure code and Failure persistent bits are user-supplied.
Failure Context Fields	
Failure code (7 bits)	Copied from bits 0:6 of `tabort` or `treclaim` source operand, set to 0 by other failure types.
Failure persistent	Indicates the failure is likely to recur on each execution of the transaction. This bit is a hint. It is copied from bit 7 of tabort. or treclaim. source operand, otherwise is set by hardware depending on failure type.
Suspended	When set to 1, the failure was recorded while the transaction was in the Suspended mode.
Privilege (2 bits)	The privilege mode of the thread at the time failure is recorded.
Failure Summary	Set to 1 when failure has been detected and failure recording has been performed. Failure recording occurs only if unset.
TFIAR Exact	The contents of the TFIAR register are precise.
ROT	Set to 1 when a ROT is initiated. Set to 0 when a non-ROT `tbegin` is executed.
Transaction Level (12 bits)	Nesting depth for the active transaction, if any. Otherwise 0 if the most recently executed transaction completed successfully, or the transaction level at which the most recently executed transaction failed.

Table 2: Transaction Exception and Status Register (TEXASR) Format. Each field is a single bit unless otherwise noted.

a special condition code being set indicating transactional "failure to launch"; the failure handler has the choice of co-opting the TM facility (via an operating system service) or choosing an alternative non-transactional path.

2.3 Rollback-only Transactions

Rollback-only transactions (ROTs) are used for single thread algorithmic speculation, and can be initiated by an extra parameter to `tbegin`. They are not used to manipulate shared data, enabling the following efficiencies as compared to atomic (conventional) transactions. First, since atomicity is not provided, ROTs are not required to do any conflict tracking. Software must ensure that ROTs do not access potentially shared data. However to provide support for rollback, ROTs must track the write set of the transaction. As a result, given a fixed-size access tracking buffer that must only track stores for ROTs, ROTs can be significantly larger than atomic transactions. Second, ROTs do not maintain the serializability of atomic transactions nor do they have the implicit barriers associated with atomic transactions (explained below in Section 4).

Just as with atomic transactions, ROTs may be nested. ROTs are nested using a flattened nesting model. ROTs may be nested with atomic transactions, but beginning with the first nested atomic transaction, the execution will be characteristic of an atomic transaction (e.g. read set tracking will be performed).

3. ROBUST ARCHITECTURAL SUPPORT FOR EXCEPTIONAL CONDITIONS

In the following section, we summarize three features of the TM architecture that qualitatively improve its capacity for supporting hardware and software corner cases: debugging and other escapes via suspended transactions, support for system interactions (interrupts, exceptions, and system calls), and support for detecting conflicts with concurrent translation shootdowns.

3.1 Explicit Suspend and Resume

While the suspend mode described in section 2.1 is largely motivated by interrupts, controlling it explicitly through `tsr` is also included for the following reasons:

- To support breadcrumb-style debugging through non-transactional Suspend mode stores.
- To support polling within a transaction, allowing commit to be dependent on external signaling, for use in speculative multithreading and helper-threading systems.
- To allow registration of commit and compensating action routines to be run at commit or abort [9, 18].
- To allow emulation routines to execute in the Transactional, rather than Suspended mode using `tresume`.

As an example of bread-crumb style debugging, an implementation of a transactional print function using the `tsr` instruction (through mnemonics `tsuspend` and `tresume`) is shown in Figure 2. Such a function can be called from within a transaction to non-transactionally record the string parameter. On transaction commit or abort, a subsequent call to a conventional printf routine is made using variable `buffer`. In this example, the `tcheck` instruction is used to detect transaction failure, terminating the string copy early. In the event of transaction failure, a partial version of the input string will be printed, which would indicate the transaction failed asynchronously in the middle of the string copy.

3.2 Interrupt and Exception Handling

One of the basic tenets of the Power ISA is the option of different implementations to choose not to implement in hardware certain instructions, or certain instruction corner cases (e.g. some rarely occurring memory alignment cases), and instead rely on software emulation of these instructions through the use of different types of interrupts (for example alignment interrupts for some unaligned accesses, or hypervisor emulation assistance interrupts for unimplemented

```
char buffer[];  //preallocated output buffer
int curindex = 0;
void trans_print(char *str) {
   int i = 0;
   tsuspend        //Suspend transaction
   //while not the end of str
   while(str[i] != 'NUL') {
      buffer[curindex] = str[i++];

      //break on failure, since last read could be garbage
      if(!tcheck) break;
      curindex++;
   }
   //terminate string with NULL char
   buffer[curindex] = 'NUL';
   tresume                  //Resume transaction
}
```

Figure 2: A transactional print routine. Assumes a conventional printf is called on transaction commit or failure using `buffer`. Input parameter `string` may be partially printed in the event of transaction failure.

instructions). When designing the transactional instruction set extensions for the Power ISA, we were faced with the choice of architecting an "abort on interrupt" model similar to Sun Rock [7] and IBM zEC12 [16][5]. Instead we chose a "Suspend on interrupt" model, in which the machine mode is changed from Transactional mode to Suspended Transactional mode on the occurrence of the interrupt. We made this decision for the following reasons:

- to allow for the flexibility of instruction emulation in future implementations without compromising the execution of enclosing transactions, for example to prevent cases where code running on one system is able to successfully use transactions, while the same code on a later generation of system suffers from transaction failure due to emulation.
- to allow for transactions to survive memory-related interrupts, for example those caused by translation faults that may miss in our hashed page table but are resident in the backing software-managed page table and can be quickly serviced by the operating system.
- to allow for transactions to survive other types of interrupts, such as those that are caused by interactive debuggers, or timer interrupts that do not result in a context switch.
- to allow for the availability of certain transaction-aware operating system services, as have been motivated by other prior work [20, 2].
- to allow future flexibility, in areas that have not yet been anticipated.

In the abstract, the suspend on interrupt model does simply that: the machine state is transitioned from a Transactional mode to a Suspended Transactional mode when interrupt occurs. The mechanics of running the operating system in the Suspended mode, and managing the register checkpoint of multiple transactions, are described in the following two subsections.

[5]Intel Haswell also appears to support an abort-on-interrupt model, although the Intel TSX architecture does not rule out future support for interrupts and system calls within RTM and HLE regions [15].

3.2.1 Mode Transition

In the Power ISA, like others, first-level interrupt handlers (FLIHs) assume a specific operating environment, for example that they will be invoked with external interrupts masked. Consequently, the process of generating an interrupt involves the implicit setting of MSR state by hardware on the occurrence of the interrupt to a FLIH-friendly state, as well as making a backup of the MSR version at the time of the interrupt, which is recorded in register SRR1. When interrupt handling is complete, an `rfid` (Return From Interrupt) instruction is used that returns control to the interrupted context, including resetting the MSR state using the contents of SRR1. With these existing mechanisms, it was natural to implement the suspend-on-interrupt mechanism by simply setting `MSR[TS]` to Suspended on any interrupt, and recording the previous value of `MSR[TS]` in SRR1 like any other mode bits.

For example, in cases where a page fault occurs during a transaction, for a page which is not in the hashed page table but is available in the software-managed backing page table, the interrupt handler is invoked in the Suspended transactional mode, and the previous version of the `MSR[TS]` field, indicating that the thread was transactional, is saved in the SRR1 register. After installing the mapping in the hashed page table, the OS will resume the interrupted thread using its conventional return-from-interrupt sequence. Since the SRR1 register indicates it was in the Transactional mode at the time of the interrupt, the thread will naturally be returned to the Transactional mode on the execution of `rfid`, allowing the transaction to continue without failure.

Because the handling of failures that occur in the Suspended mode is deferred until resuming the Transactional mode, the OS or hypervisor handling the interrupt is protected from being routed to the failure handler; execution will continue without interruption due to failure until the transaction is resumed.

3.2.2 Time-sharing the Transactional Facility

While some interrupts can be serviced without failing an interrupted transaction, in many circumstances (e.g. context switches) the transactional facility will need to be freed and re-initialized for re-use. For this purpose, we define an instruction called `treclaim` that is similar to tabort in that it reverts registers and memory to their pre-transactional state. However, it does not transfer control flow to the failure handler. For example, as shown in Figure 3, an operating system or hypervisor can use `treclaim` during processing of a timer interrupt after first saving the state of the interrupted transaction. Since the `treclaim` instruction will revert the checkpointed registers (including GPRs) to their pre-transactional state, the necessary code (not shown) resembles a first-level interrupt handler in terms of its register use. Since its GPRs are overwritten by the `treclaim` it must leverage privileged registers that are not reverted. The OS then has access to the checkpointed register state, which can also be saved.

In the sequence shown in Figure 3, the timer interrupt occurs in the middle of a suspended transaction. Due to the deferred-failure semantics of suspended transactions, both

```
tbegin.
...
tsuspend.  //Thread starts Suspend mode block
...
```

Decrementer interrupt
OS interrupt handler runs
OS chooses to switch threads
OS saves precise register state from moment of interrupt
`treclaim.  //failure recording occurs`
OS saves register state checkpointed at tbegin
OS schedules a different thread
...
OS chooses to reschedule original thread
OS restores thread's checkpointed register state
`trechkpt.  //checkpoints current regs`
OS restores thread's precise register state
`rfid //Return from interrupt`

```
...
tresume.  //Thread completes suspend-mode code
```
Hardware performs failure handling
Failure handler executes

Figure 3: Context-switch during transaction, OS returns to precise interrupt location for completion of suspend-mode code block.

precise interrupt state, as well as checkpointed state, are saved. Once the OS chooses to reschedule the interrupted thread, it restores the checkpointed register state, then resaves this state in the hardware register checkpoint using a new instruction called `trechkpt`. Once the register checkpoint is re-initialized, the OS will restore the precise interrupt state before returning to the interrupted suspend block via `rfid`. Execution continues in the suspend block until the transaction is resumed, which triggers failure handling.

3.3 Transactions and Translation Shootdown

Most architectures (the Power ISA included) guarantee that once a translation entry for a page has been removed from the page table and invalidated from translation buffers, the page is no longer accessible. However, since TM systems delay the exposure of transactional stores until commit, which is decoupled from the address translation happening as memory accesses are executed (in a conventional microarchitecture), there is a window of vulnerability to translation invalidations where (unless detected) transactions may erroneously commit despite conflicts. As an example, consider the code shown in Figure 4, in which a transaction executed by Thread 1 erroneously commits despite a conflict by Thread 3. While this example may seem implausible given most page fault handling latencies (e.g. disks), a variety of lower-latency memory-mapped devices (e.g. solid-state disks) significantly increase the likelihood. Even in the absence of such low-latency paging mechanisms, however, correctness is a requirement no matter how unlikely an error may appear to be.

Some architectures implement TLB shootdown via interprocessor interrupts [14, 29, 1], whereby the operating system will interrupt each processor in the system, causing that processor to locally execute a TLB invalidation instruction. In such systems, forcing transactions to fail on the occurrence of such interrupts, or on the occurrence of a locally executed translation shootdown instruction, is sufficient to

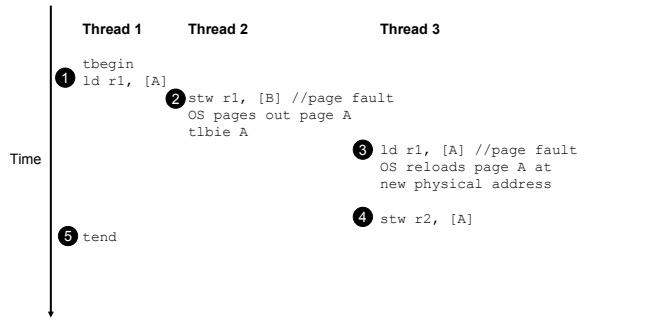

Figure 4: Interaction between transaction and TLB shootdown: (1) Thread 1 starts a transaction and loads from page A. (2) Thread 2 incurs a page fault to a different page, causing page A to be paged out. (3) Thread 3 incurs a page fault to page A, and the OS installs page A at a new physical address. (4) Thread 3 performs a store to page A, which should conflict with Thread 1's transaction. (5) Thread 1's transaction erroneously commits.

prevent the associated correctness issue.

On architectures that support TLB shootdown via invalidation instructions that result in a system-wide TLB shootdown [13, 12], including the Power ISA, such reliance on interrupts or locally executed instructions is not possible. In such architectures, a translation invalidation instruction (e.g. `tlbie`) causes a broadcast to all translation buffers, where matching entries are invalidated.

In the Power ISA, which employs a two-level virtual memory system, translation entries in first-level TLBs (called ERATs in IBM terminology) are dependent on two different resources: page table entries, which are invalidated using `tlbie` instructions that cause system-wide page-table shootdown, and segment table entries, which are invalidated using segment table invalidation instructions (e.g. slbie, slbia) that affect only local segment tables. The architectural semantics we define for the Power ISA guarantees that page table invalidations due to `tlbie` instructions executed by any thread will cause transaction failure, if the invalidation of the associated page's translation would compromise the semantics of the transaction. For segment table invalidations, we require that a programmer performing segment table invalidation also ensure transactional correctness by explicitly aborting any transaction currently executing on the thread. Such segment table invalidation cases are highly unlikely given the low frequency of segment table invalidation, but they may occur in some corner cases when a user-level transaction is interrupted.

While the details of conflict detection mechanisms between transactions and translation shootdowns are implementation-specific and therefore beyond the scope of this paper, it is important to note that `tlbie` operations have a system-wide effect and can occur at a high rate in many workloads. As such, as the number of hardware threads per system increases the frequency of system-wide `tlbie` operations will similarly increase, and simply aborting a transaction on the receipt of any translation shootdown operation will lead to frequent unnecessary transaction failures. Despite significant research in the area of transactional

memory, we are not aware of any prior work that describes this problem. Solutions include tracking the transactionally accessed pages (either by marking translation buffer entries or via a bloom-filter-like approach) and performing conflict detection with translation invalidations; we consider the evaluation of such solutions an open area of research. Others have described the need to detect interactions between transactions and translation shootdown in the context of virtualized TM systems, where the affected transaction has been context-switched out [25]. We show that this problem can also exist for active transactions given concurrent paging by other threads.

4. INTERACTION OF TM WITH THE POWER ISA MEMORY MODEL

Implementation of transactional memory in a weakly-ordered, non-multi-copy-atomic[6] architecture such as the Power ISA presents certain unique challenges. Such an architecture permits the non-atomic propagation of writes and significant reordering of memory accesses. In contrast, transactions, by their very nature, require that the accesses within a transaction be presented to the rest of the system as an indivisible atomic unit. Also, specific measures must be taken to ensure that transactional and non-transactional accesses interact in a sensible manner.

In the following sections we begin with an overview of the Power ISA memory model and then discuss how transactions are serialized in an atomic fashion over the weak memory model. We then describe a unique barrier effect needed at transaction committal to ensure the proper inter-operation of transactional and non-transactional accesses. Finally, we conclude with a discussion of how the Power ISA atomic update instructions `larx` and `stcx` interact with transactions, as well as the ordering properties of `tbegin`, `tend` and memory barriers within transactions.

4.1 Power ISA Memory Model Background

While a complete description of the Power ISA memory model is beyond the scope of this paper, the following brief summary will provide some context for the subsequent discussion of the interaction of TM with weak ordering. More complete descriptions are available elsewhere [12, 27, 26].

Unlike stronger memory models such as Sequential Consistency and TSO, the Power ISA memory model does not guarantee multi-copy atomicity of writes or ordering among accesses to differing addresses. When a program requires ordering or atomicity, the programmer must use dependencies, synchronization (`isync`), or memory barriers like `sync` and `lwsync`. The `sync` instruction provides the most stringent ordering guarantee, enforcing order and atomicity for all four possible ordered pairs of access types: *load* $\rightarrow$ *load*, *load* $\rightarrow$ *store*, *store* $\rightarrow$ *load*, *store* $\rightarrow$ *store*. The lightweight sync (`lwsync`) instruction provides the same ordering guarantees as `hwsync` except the *store* $\rightarrow$ *load* ordering (i.e. loads following a `lwsync` may be performed with respect to other processors before stores that precede the `lwsync`). The `sync` and `lwsync` instructions provide a *cumulative* memory barrier that additionally and transitively orders all applicable memory accesses that have been performed with respect to the processor executing the memory barrier before all memory accesses caused by the instructions that follow the memory barrier. The cumulative memory barrier also additionally and transitively orders, after the memory barrier, all applicable memory accesses, by other processors, that are performed after that other processor has read a value that is already ordered after the memory barrier.

4.2 Serialization of Transactions

The TM architectue extensions to the Power ISA define a property of serializability of transactions which states that successful transactions are serialized in some order and that no processor (whether using transactional or non-transaction loads and stores) can observe the accesses caused by these transactions as occurring in an order conflicting with the serialization order. To achieve this serialization, we follow principles similar to those for the existing Power ISA atomic update instructions: Load and Reserve (`larx`) and Store Conditional (`stcx`). `larx/stcx` instruction pairs provide a means to perform an atomic update of a memory location even in the absence of multi-copy atomicity.

Unlike in earlier non-multi-copy memory models, the non-atomic propagation of writes in the Power ISA leads to certain unique requirements for processing memory updates to allow `larx/stcx` pairs to produce an atomic update. These requirements are insufficient for transactions and must be extended to allow transactions to serialize properly as discussed below. A `larx` instruction may read a stale value for a location (a value that is not the most recent value serialized in the coherence order for the location), but if it does, the `larx/stcx` pair will fail and will not update memory. If, however, the `larx` reads a non-stale value and the subsequent `stcx` to the same location can place its written value immediately after the value read by the `larx` in the coherence order for the location, the `larx/stcx` pair succeeds and updates memory. [12, 27, 26]. A `stcx` instruction can succeed as soon as its write is serialized into the coherence order, but before the effects of the write have propagated to the other processors. In other words, other processors may still read a stale value for a location that has been written by a `stcx`. The effect of an atomic update is maintained, however, because all threads atomically updating the location must use `larx/stcx` instruction pairs and if a competing `larx` reads a stale value, the competing thread's `larx/stcx` pair will fail.

Transactions provide their serialized atomic updates in a similar fashion. To provide this effect, transactional loads, like `larx` instructions, must read a non-stale value or the transaction will ultimately fail. In addition to reading a non-stale value, the value read by a transactional load must still be the most recent value in the coherence order for the location when the transaction reaches the final `tend` instruction and the transaction must complete before any new value is serialized into the coherence order for the location, otherwise the transaction will fail. In contrast, a transactional store must gain control of the coherence mechanism for the given location to ensure that the transactional write will enter the coherence order for the location as the most recent value, and that no store by another thread will subsequently be serialized into the coherence order for that location before the transaction commits. This is typically achieved by gaining write ownership of the location in the coherence protocol and either holding that ownership until the transaction commits successfully or failing to hold that ownership due to a conflict with another transactional or non-transactional access and failing the transaction.

Thread 1	Thread 2	Thread 3	Thread 4
tbegin.	ld X	ld Y	tbegin.
std X=1	<addr>	<addr>	std Y=1
tend.	ld Y	ld X	tend.

Figure 5: Independent Reads of Independent Writes (IRIW). All memory locations and registers initially contain a value of 0.

Thread 1	Thread 2	Thread 3
std X=1	tbegin.	ld Y
	ld r1, X	<addr>
	std r1, Y	ld X
	tend.	

Figure 6: Example of cumulative memory access ordering

Additionally, a major difference between transactional semantics and `larx/stcx` semantics is that, unlike `larx/stcx` pairs for which the `stcx` can succeed once the write has been serialized into the coherence order and before the write has propagated to all processors to eliminate stale copies, the transaction must propagate (or at least appear to propagate) to all processors and make any stale cached copies of the locations un-readable before a transaction can successfully commit. This is necessary to prevent non-transactional loads on other threads observing an inconsistent image of the aggregate store.

To illustrate why the aggregate store must logically propagate to all processors before the transaction commits, consider the transactional variation of the well known IRIW testcase [4] illustrated in Figure 5. In this testcase, threads 1 and 4 are transactions consisting of a single store of '1' to X and Y respectively (these code examples are illustrated in pseudo-code form and omit the failure handler branches for clarity). By serialization, all threads, whether using transactional or non-transactional accesses, must observe the transactions as occurring in the same order. Both threads 2 and 3 read the locations written by threads 1 and 4, but in a reversed order (the order of the loads on thread 2 and 3 are maintained by an address dependency from the first load's returned value to the second load's address illustrated as <*addr*> in the figure). A serialization error would occur if both threads read a value of '1' for their first read and a value of '0' for their second read.

If the transactions were able to commit when their writes had been serialized into the coherence order but before those writes had propagated to all the threads, the write propagations could happen in different orders at threads 2 and 3 and lead to a serialization error. Transactional writes propagating to all threads and rendering all stale copies of the locations unreadable before the transaction can commit prevents this serialization error. It should be noted that aggressive implementation techniques can be employed to achieve the net effect of a full propagation without actually requiring the full propagation to occur before the transaction can commit.

It is interesting to note that if all accesses to locations were constrained to be transactional accesses, a transaction could commit as soon as the writes in the aggregate store were serialized into the coherence order. The transactional loads on other threads would, much like `larx`/`stcx` pairs, cause the transaction to fail if a stale value was read. However, locations written by transactions can be read by non-transactional loads and maintaining serialization requires that the aggregate store propagate to all threads, to render stale values unreadable, before the transaction commits. In essence, the requirement to propagate the transactional stores fully before committing the transaction is necessary only to support non-transactional accesses interacting with transactional accesses in a sensible manner.

To summarize, for a transaction to commit successfully, all transactional loads must read non-stale values and these values must remain the current values until the transaction commits. All transactional stores must gain control of the coherence mechanism and ensure that no other stores can serialize into the coherence order for the location until the transaction successfully commits, and furthermore the effects of all transactional stores must propagate (or appear to propagate) to all threads to ensure that no thread can read, either transactionally or non-transactionally, any stale value for those locations before the transaction can successfully commit.

4.3 Non-transactional Access Cumulativity

Another interaction between non-transactional accesses and transactional accesses relates to cumulativity as illustrated in Figure 6. In this testcase, thread 1 non-transactionally stores the value '1' to location X. This store propagates and becomes visible to the threads 2 and 3 at different times (due to the absence of multi-copy atomicity). Thread 2 reads the new value for X and stores that value into Y within a transaction. Finally, thread 3 non-transactionally reads the new value of Y and then (due to a address dependency illustrated as <addr> in the figure) reads X. Should the read of X by thread 3 return the new value of X stored by thread 1? Presumably yes, because the programmer would not expect the third thread to see the new value of Y from the second thread, which was set to the valued stored to X by the first thread, and then not see the new value of X by reading X directly. Causality would be violated: an effect of the store to X, namely the new value in Y, was visible to thread 3 before the cause, the store to X itself, was visible.

However, the process of committing the transaction on the second thread, as described above, does not ensure that thread 3's read of X will return the value stored by thread 1. Committing the transaction on the second thread does not affect the propagation of non-transactional stores by other threads (thread 1 in this case) that the transaction may have read. In this example the non-transactional store by thread 1 needs to be ordered ahead of the transaction's aggregate store, at each thread, to ensure that any stores the transaction read propagate to third party threads (in this case thread 3) before the aggregate store from the transaction.

To achieve this ordering, the architecture requires a `tend` instruction that ends a successful transaction to create an implicit memory barrier. This memory barrier, called

the integrated cumulative memory barrier, orders all non-transactional accesses, by other threads, that are performed with respect to the transaction's thread before the barrier is created before the aggregate store, and before all non-transactional accesses, by other threads, that are performed after the other thread has read a value stored by the aggregate store or by any non-transactional store that is already ordered after the barrier.

In the example above, the integrated cumulative memory barrier causes thread 1's store to X to be ordered before the transaction's aggregate store to Y at thread 3. This ensures that thread 3 feels the effects of any memory accesses that affected the transaction (thread 1's store to X in this example). The effects of this barrier are similar to the cumulative effect of the normal memory barriers in the Power ISA, but are limited to non-transactional accesses by other threads and the aggregate store of the transaction.

The integrated cumulative memory barrier is defined to order only non-transactional accesses before the barrier because, due to serialization, transactional accesses are fully propagated before a transaction can commit. Thus transactional accesses do not need to be "pushed along" by the integrated cumulative memory barrier. Once again, this is a circumstance in which specific architectural mechanisms were required in order to ensure a sensible interaction between transactional and non-transactional accesses.

4.4 Interactions with Load And Reserve and Store Conditional

In most cases, `larx`/`stcx` pairs that straddle transactional mode changing instructions lead to nonsensical programming constructs. To simplify implementations and verification, the TM architecture extensions to the Power ISA require that the reservation is reset when crossing over these mode changing instructions with one exception. The exception is that, within a transaction, if a `larx/stcx` pair straddles a `tsuspend/tresume` region (i.e. `tbegin/larx/tsuspend/tresume/stcx/tend`), the reservation is not reset by `tsuspend` and `tresume`. This is to permit emulated instructions (which are handled by an interrupt which causes an implicit suspend region to occur while the instruction is emulated) to be used within a `larx/stcx` pair within a transaction. In all other cases, the reservation is reset. This includes a `larx`/`stcx` pair that completely straddles an entire transaction. It was not deemed necessary to support a transaction executing inside a `larx`/`stcx` pair and not supporting it reduced verification complexity. In general, for a `larx/stcx` pair to succeed, the `larx/stcx` pair must be either (a) entirely outside of a transaction and not straddling a transaction or (b) within a single transaction, but possibly spanning over suspend region(s) within that transaction.

With the exception of the reservation flag resets mentioned above, the TM extensions to the Power ISA explicitly allows for independent, decoupled, implementations of the `larx/stcx` and transactional mechanisms. For example, we explicitly allow for, and implementations will exploit, the possibility that a `larx/stcx` pair inside a transaction fails, but the enclosing transaction still succeeds. This is to allow for certain implementation conditions that need to reset the reservation flag speculatively, and thus cause the `stcx` to fail, but don't need to kill the transaction. The architecture does not, however, preclude implementations in which the `larx/stcx` and transactional mechanisms are implemented as a single unified mechanism. In either style of implementation, a `larx/stcx` pair inside a transaction may succeed as far as is necessary to honor the semantics of `larx/stcx` (e.g. the `stcx` may succeed when the write is serialized into the coherence order, but before the write has fully propagated). However, the `stcx` is still transactional, and cannot become visible to other threads until the conditions necessary for the enclosing transaction to commit are met. In other words, a `stcx` may succeed within a transaction, and the enclosing transaction may fail, in which case the `stcx's` store will not be seen by other threads. Also, while transacations can be used to perfrom simple atomic updates on single locations in memory, `larx`/`stcx` pairs are more efficient for these updates and should be used instead of transactions.

4.5 Memory Barriers and Transactions

Since one motivation for TM is to simplify multi-threaded programming in the Power ISA, providing a TM semantics in which the aggregate store is not ordered with respect to the surrounding code would run counter to our goal, even though such semantics would be consistent with the philosophy of weak ordering. For this reason, and because we could implement them efficiently, we defined additional implicit memory barriers (beyond the integrated cumulative memory barrier described above) which are created by the outermost `tbegin` and `tend` for a successful transaction. These implicit barriers act like `sync` barriers placed before and after the transaction with the exception that, unlike `sync`, they do not order non-cacheable accesses. In a sufficiently aggressive implementation, implementation techniques largely eliminate the cost of these implicit memory barriers. Because the memory accesses within the transaction (in particular the transactional stores) are conditional on the transaction succeeding, the processing required to honor the implicit memory barrier effects for `tbegin` can be deferred until `tend` (or until entering the first suspend region at which point a barrier is created to honor the ordering effects of the `tbegin` barrier). The processing required to successfully commit a transaction is essentially the same as the processing necessary to enforce the implicit barrier for both `tbegin` and `tend` instructions and therefore the cost of implementing these barriers is essentially overlapped with the `tend` processing costs. The implicit barriers are therefore far more efficient than explicit `sync` instructions placed before and after the transaction. While, as a consequence of the implicit barriers, an empty transaction (one with no loads or stores) that succeeds will have the same barrier effects as a `sync` for cacheable operations, it is more efficient to use a `sync` instruction explicitly. Because the implicit memory barriers had a low performance cost, benefited important use cases such as transactional lock elision in Java, and simplified the programming model, we included them in the `tbegin`/`tend` behavior. If future circumstances warrant, a "weak" `tbegin` and `tend` without these barriers can be added to the architecture.

Ignoring suspend/resume regions, it is generally not useful to employ explicit memory barriers within a transaction because the loads and the stores within a transaction logically happen as an indivisible atomic unit, so there is no relative ordering that a barrier inside a transaction can effect for these accesses. However, an explicit memory barrier within a transaction between two suspend regions will order

the sets of accesses within those two suspend regions. Also, explicit memory barriers within a suspend region will order the accesses within that suspend region and also accesses within other suspend regions straddling that suspend region. Generally speaking, barrier instructions occurring anywhere provide their usual ordering effects for non-transactional or suspend mode accesses straddling the barrier instruction.

These rules were intended to allow processors to implement the explicit barrier instructions in an "as-they-come" fashion. The processors simply implement the barriers using the existing mechanisms. This eases implementation and verification complexity. These rules also ease the transactionalization of existing code. A barrier within code that is being transactionalized retains its usual ordering properties for accesses not included in the transaction. While the implicit `tbegin/tend` barriers would enforce most orderings done by a barrier within a transaction, the implicit barriers do not order non-cacheable operations and therefore the explicit barrier must provide this ordering even after it is included in a transaction. Explicit memory barriers within the transaction should occur only when existing code is being transactionalized. New code should avoid explicit memory barriers within the non-suspend regions of transactions as they serve no useful purpose that cannot be met by using memory barriers outside the transaction.

5. OTHER RELATED WORK

Recent work in defining commercial ISA support for TM, including our own, builds upon significant prior work that appears in the literature, more than there is space to discuss here. We will focus on prior work that is particularly relevant to the novel features described.

Prior work has proposed transactional pause/unpause operations [30], pause/resume[25], xpush/xpop [11], and escape actions [20], and open nesting mechanisms [18] that provide support for transaction suspension, including interaction with operating system services during transactional execution. A significant complication of any form of transactional suspension is the handling of the transactional register checkpoint, which must be saved/restored in the presence of context switching. While prior work in the area has assumed that such register checkpoints would be saved/restored by software, the interfaces for accomplishing the access of checkpointed registers were not explicitly discussed. The addition of `treclaim` and `trechkpt` primitives enable such suspend mode regions to be multiprogrammed. While transaction suspension can be used to support interrupts, I/O, and system calls in a transaction, others have proposed such support via various forms of transaction inevitability [8, 3]. Chung et al. explored the handling of interrupts in transactions, selecting among policies for delaying interrupt processing, aborting, or context-switching a transaction depending on interrupt type[5]. Our suspend-on-interrupt approach allows for software-defined behavior tailored to each type of exception, interrupt, or system call, including interrupts due to emulation.

At this point in time, HTM has officially been adopted in three commercial architectures: x86-64, Power ISA, and the IBM z/Architecture®. Although there are numerous minor differences among the three, major differences include the inclusion of Hardware Lock Elision (HLE) in x86-64, constrained transactions in the z/Architecture, and Suspend/Resume (including suspend on interrupt) and ROTs in the Power ISA.

In lieu of supporting a general suspended transactional mode of execution, the z/Architecture developers include a new opcode for explicit non-transactional stores, providing part of suspend mode's value while avoiding some of its design complexity (for example its support for deferral of transaction failure handling). We also considered such an approach; however in addition to the motivations listed in Section 3 for suspend/resume, there was also a fear that over time piecemeal support for non-transactional operations within transactions would proliferate on a case-by-case basis, particularly considering TM's early but evolving software use cases. While we saw an immediate value in suspend/resume for the aforementioned reasons, we also decided the investment in suspend/resume may have unforeseen benefits in the future as TM use cases evolve, since it can be used to enable software implementations of many features that we do not support in hardware.

Support for constrained transactions in the Power ISA was also given serious consideration, however the burden of implementing and verifying such guarantees, particularly the guarantee of a minimum transactional data footprint per thread in the presence of multithreaded core designs and shared caches, precluded adoption of the feature.

The inclusion of HLE is a notable difference between x86-64 and both the Power ISA and z/Architecture. This feature decreases the barrier to entry for adopting lock elision in some applications, particularly those that might include custom-implemented lock-routines (where lock-elison enabled versions of the synchronization library may not be commonly available) by maintaining the illusion of a "locked" lock to code in the critical section. We note that many prevalent programming languages and synchronization libraries (e.g. Java, C/C++ with libpthreads and openmp) do not expose the lockword location or its value semantics to the end-user, and consequently maintaining the appearance of a "locked" lock to that end-user code is unnecessary in these environments; therefore, unmodified applications can benefit from lock elision through support in libraries or runtimes.

6. CONCLUSIONS

The work to include TM in the Power ISA continues through current implementation efforts to be released in future pSeries systems. This multi-year architectural definition process weighed the needs and opinions of many individuals across a wide range of stakeholders: application developers, JVM and C/C++ compiler writers, operating system (AIX®, Linux, IBM i) and hypervisor developers, as well as hardware designers responsible for its implementation and verification. While many aspects of the resulting architecture will look familiar to those following the research literature, we were surprised by the number of previously undescribed interactions with existing architectural features. Our goal has been to develop an enterprise-class TM architecture capable of handling these, and other, surprises in the context of an extremely permissive memory model, to serve as the basis for new generations of Power® systems.

Acknowledgments

The definition of this architecture was made possible by contributions from a wide group of IBMers. We would especially like to acknowledge Bob Blainey, Mary Brown, Susan Eisen,

Guy Guthrie, Tom Heller, Ben Herrenschmidt, John Ludden, Paul MacKerras, Paul McKenney, Steve Monroe, Jose Moreira, Naresh Nayar, DQ Nguyen, Randy Pratt, Pratap Pattnaik, Raul Silvera, Bill Starke, Randy Swanberg, Jim Van Norstrand, and Julian Wang. We would also like to thank Christos Kozyrakis, and our reviewers for their feedback on this work.

7. REFERENCES

[1] *ARM Arch. Reference Manual*. ARM Ltd., 2005.

[2] L. Baugh and C. Zilles. An analysis of I/O and syscalls in critical sections and their implications for transactional memory. In *Proc. of the 2008 Intl. Symp. on Performance Analysis of Systems and Software*.

[3] C. Blundell, E. C. Lewis, and M. M. K. Martin. Unrestricted transactional memory: Supporting I/O and system calls within transactions. Technical Report CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, Apr 2006.

[4] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *Proc. of the 2008 Conf. on Programming language design and implementation*, 2008.

[5] J. Chung, C. C. Minh, A. McDonald, T. Skare, H. Chafi, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. Tradeoffs in transactional memory virtualization. In *Proc. of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2006.

[6] W. W. Collier. *Reasoning about parallel architectures*. Prentice-Hall, Inc., 1992.

[7] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proc. of the 14th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[8] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proc. of the 31st Intl. Symp. on Computer Architecture*. Jun 2004.

[9] T. Harris. Exceptions and side-effects in atomic blocks. *Sci. Comput. Program.*, 58(3), Dec. 2005.

[10] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proc. of the 20th Intl. Symp. on Computer Architecture*, 1993.

[11] O. S. Hofmann, D. E. Porter, C. J. Rossbach, H. E. Ramadan, and E. Witchel. Solving difficult HTM problems without difficult hardware. In *TRANSACT '07: 2nd Workshop on Transactional Computing*, August 2007.

[12] IBM Corporation. *Power Instruction Set Architecture v.2.06B*, July 2010.

[13] Intel Corporation. *Intel IA-64 Architecture Software Developers Manua l, Volume 2: IA-64 System Architecture, Revision 1.1*, July 2000.

[14] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, August 2012.

[15] Intel Corporation. *Intel Architecture Instruction Set Extensions Programming Reference: Chapter 8: Intel Transactional Synchronization Extensions*, Feb. 2012.

[16] C. Jacobi, T. Slegel, and D. Greiner. Transactional memory architecture and implementation for IBM System z. In *Proc. of the 45th Intl. Symposium on Microarchitecture*, December 2012.

[17] M. M. K. Martin, C. Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2), 2006.

[18] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *Proc. of the 33rd Intl. Symposium on Computer Architecture*, 2006.

[19] M. Moir, K. Moore, and D. Nussbaum. The adaptive transactional memory test platform: A tool for experimenting with transactional code for Rock. In *TRANSACT '08: 3rd Workshop on Transactional Computing*, February 2008.

[20] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in LogTM. In *Proc. of the 12th Intl. Conf. on Architectural support for programming languages and operating systems*, 2006.

[21] T. Nakaike and M. M. Michael. Lock elision for read-only critical sections in java. In *Proc. of the 2010 Conf. on Programming Language Design and Implementation*.

[22] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware atomicity for reliable software speculation. In *Proc. of the 34th Intl. Symposium on Computer Architecture*, 2007.

[23] S. J. Patel and S. S. Lumetta. rePLay: A hardware framework for dynamic optimization. *IEEE Transactions on Computer Systems*, 50(6), June 2001.

[24] R. Rajwar and J. R. Goodman. Speculative lock elision: enabling highly concurrent multi-threaded execution. In *Proc. of the 34th Intl. Symposium on Microarchitecture*, 2001.

[25] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proc. of the 32nd Intl. Symp. on Computer Architecture*, 2005.

[26] S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams. Synchronising C/C++ and POWER. In *Proc. of the 33rd Conf. on Programming Language Design and Implementation*, 2012.

[27] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *Proc. of the 32nd Conf. on Programming Language Design and Implementation*, 2011.

[28] L. Su and M. H. Lipasti. Speculative optimization using hardware-monitored guarded regions for java virtual machines. In *Proc. of the 3rd Intl. Conf. on Virtual execution environments*, 2007.

[29] D. L. Weaver and T. Germond, editors. *SPARC Architecture Manual V9)*. PTR Prentice Hall, 1994.

[30] C. Zilles and L. Baugh. Extending hardware transactional memory to support nonbusy waiting and nontransactional actions. In *Proc. of the First Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. June 2006.